Published in Image Processing On Line on YYYY–MM–DD.
ISSN 2105–1232 © YYYY IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
http://dev.ipol.im/~facciolo/ipol_demo/f_intimage_block_matching/

PREPRINT January 23, 2013

Integral Images for Block Matching

Gabriele Facciolo, Nicolas Limare, Enric Meinhardt

Abstract

The *integral image* representation is a remarkable idea that permits to evaluate the sum of image values over rectangular regions of the image with four operations, regardless of the size of the region. It was first proposed under the name of *summed area table* in the computer graphics community by Crow'84, in order to efficiently filter texture maps. It was later popularized in the computer vision community by Viola & Jones'04 with its use in their real-time object detection framework. In this article we describe the integral image algorithm and study its application in the context of block matching. We investigate tradeoffs and the limits of the performance gain with respect to exhaustive block matching.

Source Code

The source code, the code documentation, and the online demo are accessible at the IPOL web part of this article[1].

1 Introduction

The integral image representation was introduced [9, 4] with the purpose of evaluating sums of image values over axis aligned rectangular regions in constant time. Crow [4] applied it in graphics (under the name of *summed area table*) for efficiently computing *mipmaps* [24] by filtering the images with spatially varying rectangular filters. Lewis [15] used integral images for accelerating template matching. Viola and Jones [22] popularized it in the computer vision community with its application in their *real-time object detection framework*. More recently, Bay et al. [1] took advantage of integral images for the purpose of rapidly computing approximations of SIFT features with their Speeded Up Robust Features (SURF).

Block-matching aims at identifying corresponding blocks (or patches) between two images. A patch is a rectangular portion of an image. For each pixel of the first image (which we shall call from now on reference image) we need to compute the *matching cost* (denoted also *patch distance*) between the patch centered at this reference pixel, and all the patches of the second image within a given range around the reference pixel. For each patch in the reference image, the patch in the second image yielding the minimum matching cost is called *"the corresponding block"*. A popular matching cost is computed as the sum of squared differences (SSD) of pixel values within the patch. The correspondences and matching costs provide valuable information about the similarity between the patches, which can be applied to image denoising [3], video compression [14], motion and depth estimation [19] among others. One of the first uses of integral image applied to block matching

[1] http://dev.ipol.im/~facciolo/ipol_demo/f_intimage_block_matching/

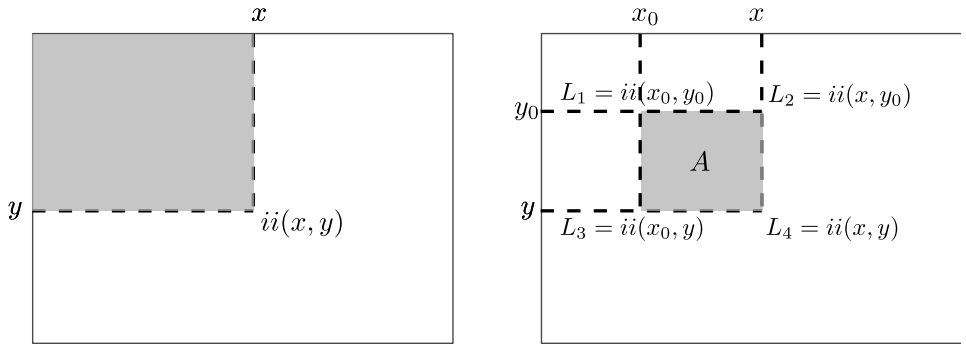

Figure 1: **Integral image and evaluation over a rectangle.** The diagram illustrates the computation of the integral image: the value of $ii(x, y)$ is the sum of the pixels in the gray region, where the upper left corner of the image is the origin of the image's coordinate system. The diagram on the right shows how to compute the sum of the samples of the discrete image on a rectangle with integer bounds, $A = [x_0, x] \times [y_0, y]$ using the integral image ii. It is enough to obtain the values of the integral image at $L_1 = ii(x_0 - 1, y_0 - 1)$, $L_4 = ii(x, y)$, $L_3 = ii(x_0 - 1, y)$ and $L_2 = ii(x, y_0 - 1)$, and to compute $L_4 + L_1 - (L_2 + L_3)$.

was in the work by Faugeras et al. [7]. In that work the authors describe an optimization to spare redundant computations, that is much in the spirit of the integral images. Later, Veksel [21] applied the integral images to stereo block matching for efficiently computing matching costs with multiple window sizes, and Wang et al. [23] used integral images for accelerating the patch comparison in the NL-means denoising algorithm [3].

The naive evaluation of the matching cost for a set of M relative offsets, using patches of size r^2, entails $O(Mr^2)$ operations for each patch in the reference image. Therefore, for an image with N pixels, the naive computation of all the matching costs requires $O(MNr^2)$ operations. Note that the patch size affects directly the computational cost. Many techniques have been proposed to reduce this computational load by avoiding unnecessary comparisons, for instance exploiting the regularity of the offset fields [18]. Covering all these techniques is well beyond the scope of this work. Here we are going to describe how the integral image can be applied to compute the matching costs with $O(MN)$ operations. Thus, the cost is proportional to the number of offsets only, regardless of the window size. This performance gain comes at the expense of some flexibility, a tradeoff that will be discussed in detail.

We start by describing the integral image computation and its evaluation. In Section 2, in Section 3 we explain how to apply the integral image for block matching and comment some issues. Section 4 concludes by highlighting the benefits and limitations of the integral image applied to block matching.

2 Algorithm for Computing Integral Images

For an image i its integral image ii at the pixel (x, y) is defined as the sum of the pixel values of i above and to the left of (x, y), namely

$$ii(x, y) = \sum_{p \leq x \, , q \leq y} i(p, q). \tag{1}$$

Given the integral image, the sum of all the pixels within a rectangle aligned with the image axes can be evaluated with four array references, regardless of the size of the rectangle. This is illustrated in Figure 1, where the sum over the rectangle $[x_0, x] \times [y_0, y]$ is computed as $L_4 + L_1 - (L_2 + L_3)$, with $L_1 = ii(x_0 - 1, y_0 - 1)$, $L_4 = ii(x, y)$, $L_3 = ii(x_0 - 1, y)$ and $L_2 = ii(x, y_0 - 1)$.

In [22] the authors describe an algorithm for computing the integral image in one-pass over the image by using the recurrence relation

$$s(x, y) = s(x, y-1) + i(x, y)$$
$$ii(x, y) = ii(x-1, y) + s(x, y),$$

where $s(x, y)$ is the cumulated row sum, and $s(x, -1) = ii(-1, y) = 0$. Algorithm 1 details the implementation of this recurrence, there the image s is replaced with two vectors for storing the row sum for the current and the previous rows. Algorithm 2 details how to compute the sum of a rectangular region of the image by evaluating the integral image.

Algorithm 1: Computing *Integral Image*.

input: An image of size $nx \times ny$: i
output: The integral image of size $nx \times ny$: ii

```
1  s(0...nx-1) ← 0;                    /* vectors of length nx initialized to 0    */
2  sprev(0...nx-1) ← 0 ;
3  for y ← 0 to ny-1 do
4      for x ← 0 to nx-1 do
5          s(x) ← sprev(x) + i(x, y);
6          if x = 0, then
7              ii(x, y) ← s(x);
8          else
9              ii(x, y) ← ii(x-1, y) + s(x);
10     swap (s, sprev);
```

Algorithm 2: Summing the pixel values in a rectangle of an image using the *Integral Image*.

input: The integral image for an image i: ii. Position, width (w) and height (h) of a rectangle contained in the image: $[x_0, x_0 + w - 1] \times [y_0, y_0 + h - 1]$.
output: Sum of the pixel values of i in the rectangle of size $w \times h$: S

```
1  S ← ii(x + w, y + h);
2  if x0 > 0, then S ← S - ii(x0 - 1, y0 - 1 + h);
3  if y0 > 0, then S ← S - ii(x0 - 1 + w, y0 - 1);
4  if x0 > 0 and y0 > 0, then S ← S + ii(x0 - 1, y0 - 1);
```

2.1 Interpretation

The simplest interpretation of the integral image comes from considering a 1D discrete signal l of length N. Any sum over an interval of the signal l (for instance from x_0 to x_1) rewrites as

$$\sum_{j=x_0}^{x_1} l(j) = \sum_{j=0}^{x_1} l(j) - \sum_{j=0}^{x_0-1} l(j) = ll(x_1) - ll(x_0 - 1),$$

where $ll(x) = \sum_{i=0}^{x} l(x)$ is the "integral line". Then given ll, any sum over an interval of l can be computed with two lookups of ll. Figure 2 illustrates how a 1D discrete signal can be interpreted as a piecewise constant function and the corresponding integral line. Note that, with this interpretation the linear interpolation of the integral image yields the integral of the piecewise constant signal.

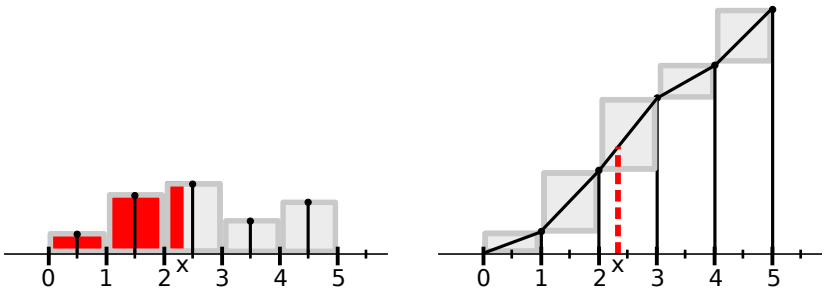

Figure 2: **Interpolating the Integral Image.** On the left it is depicted a row of pixels, while on the right it is shown the associated integral line. Note, on the left image, that the samples are interpolated with nearest neighbor and that the signal is displaced half-pixel to the right. With this convention the integral image at an integer position coincides with the integral of the piecewise constant interpolation of the samples (for instance the integral image at 2 accumulates the areas of the first and second pixels). This also implies that linearly interpolating the integral image yields the integral of the piecewise constant interpolation of the samples.

In two dimensions the integral image ii at any position (x, y) is the sum of all the pixels in the rectangular region above and to the left of (x, y) as illustrated in Figure 1. The orientation of the summed rectangle determines the "privileged orientation" for evaluating the integral image. That is, the sums over any other rectangle with sides parallel to the image axes can be evaluated with four lookups of ii, but for rectangles non parallel to the axes the evaluation of the integral image has a cost proportional to its perimeter. For 45° oriented rectangles, Lienhart and Maydt [16] proposed a rotated integral image. The rotated integral image is computed traversing the image in the diagonal directions. The sums over rotated rectangles are then computed, like the integral image, with four table lookups.

By regarding the sum of values of a signal f as a convolution with a box filter g we have that any invertible linear operator can be applied to f if its inverse is applied to g (for commodity we switch to continuous notation). Concretely with the derivative/integral pair we have (in 1D)

$$(f * g)(x) = \int f(t)g(x-t)dt = \int F'(t)g(x-t)dt = \int F(t)g'(x-t)dt,$$

where $F(t) = \int_{-\infty}^{t} f(t')dt'$. Since the derivative of the box filter yields two delta functions at the boundaries of the interval, then the computation of the sum reduces to evaluating F at the location of the deltas g'. This insight permitted the authors of [20] to efficiently filter signals with piecewise constant filters g, by converting g into a sequence of deltas at its discontinuities. The same computation extends to two (and more) dimensions by taking partial derivatives and integrals on each dimension:

$$(f * g)(x, y) = \iint F(t, s)(\frac{d}{dy}\frac{d}{dx}g(x-t, y-s))dtds, \qquad \text{with} \qquad F(t, s) = \int_{-\infty}^{s}\int_{-\infty}^{t} f(t', s')dt'ds'.$$

2.2 Generalized Integral Images

An alternative proof of the previous property [11] expresses the integration and derivation using the Fourier transform $f(x) \longleftrightarrow \hat{f}(\omega)$. The spatial integration and derivation of a function correspond to division and multiplication by $i\omega$ in the frequency domain, that is: $\frac{df}{dx}(x) \longleftrightarrow i\omega\hat{f}(\omega)$ and $\int f(x) \longleftrightarrow \hat{f}(\omega)/(i\omega)$. From that we can write

$$f(x) * g(x) \longleftrightarrow \hat{f}(\omega)\hat{g}(\omega) = \frac{\hat{f}(\omega)}{(i\omega)^n}\,(i\omega)^n\hat{g}(\omega) \longleftrightarrow \left(\int^n f\right)(x) * \left(\frac{d^n g}{dx^n}\right)(x),$$

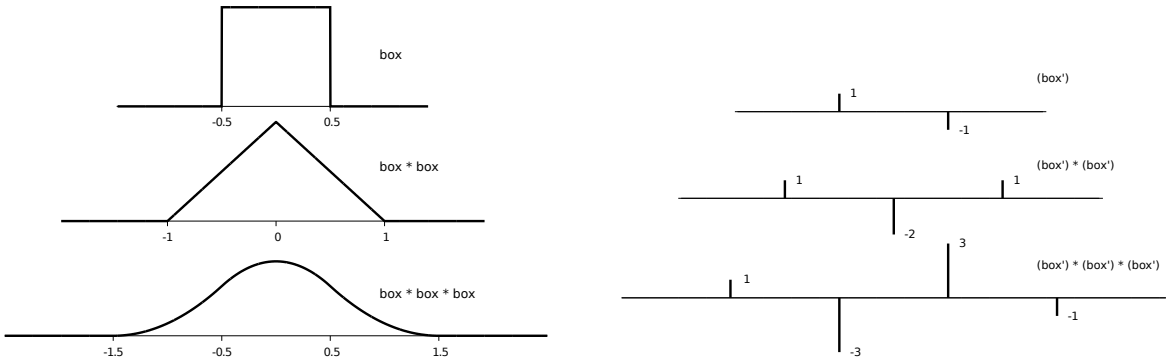

Figure 3: **B-splines of order n.** The order n B-spline is obtained by convolving the box filter n times with itself, here shown for $n = \{1, 2, 3\}$. The n-th derivative of an order n B-spline is a sequence of deltas depicted on the right. Note that the B-spline of order 3 is a good approximation of the Gaussian.

n=0	n=1	n=2	n=3
1	-1 1 1 -1	1 -2 1 -2 4 -2 1 -2 1	-1 3 -3 1 3 -9 9 -3 -3 9 -9 3 1 -3 3 -1

Figure 4: **Coefficients for 2D B-spline filtering using generalized integral images.** The 2D coefficients for filtering with a n-th order B-splines evaluating the corresponding n-fold integral image. Adjusting the spacing between the samples yields the filters of different width. The zero order corresponds to sampling the image with a delta.

where $\int^n f$ and $\frac{d^n g}{dx^n}$ denote respectively the n-fold integration and n-th order derivative of the signals.

This observation leads to a generalization of the integral images [11, 8, 6] to n-fold integral images, which permits the filtering of signals by piecewise polynomial filters in constant time, regardless of the filter's support. By increasing the integral order it is possible to define smoother and more isotropic filters, which reduce the artifacts common to the box filter.

Concretely, convolving the box filter n times with itself generates the n-th order B-spline [11, 6], whose n-th derivative is a sequence of deltas as shown in Figure 3. Therefore, given the n-fold integral of an image it is possible to evaluate its convolution with a B-spline of the same order, by evaluating some samples. In Figure 4 the coefficients for the 2D B-splines of low order are shown, adjusting the spacing between the samples yields the filters of different width.

The main limitation of the n-fold integration is the precision needed to store the integral image values. That is, storing the n-fold integral image of a signal of length 2^m and with b bits per sample, demands an integral image with a precision of $m*n+b$ bits. Another issue, to be taken into account, involves the evaluation cost associated to the generalized integral images. For evaluating an order 2 B-spline filter are needed 16 accesses to the integral image (see Figure 4), four times the amount needed for the order 1 B-spline. This implies that for small blocks (smaller than 7×7, according to Table 1) it is more effective to compute the sum without using the integral image. For more details about the generalized integral images we recommend [4, 11, 6] as further reading.

2.3 Numerical Precision Notes

The numerical precision of the integral image depends on the concrete data being accumulated, and the precision of the data type used to store the image. In this section we analyze the precision limits

of the integral image representation for some typic cases.

- **Integer pixel values.** The simplest case corresponds to the accumulation of b-bit integers into an m-bit data type. In this case one can accurately accumulate at most $2^{(m-b)}$ values without precision loss.

  As an example, if the data vector contains 12-bit integers and the integral image is stored as a single precision float (32-bit, IEEE754) which has a 24 digits mantissa, then only $2^{(24-12)} = 4096$ values can be accumulated before small ones are at risk of being lost. While a double precision integral image, which has 53 digits (64-bit float, IEEE754), can hold the much reassuring quantity of 2.19×10^{12} values.

- **Fixed point pixel values.** Usually we consider data with fixed point values of the order of the square of the underlying image pixels. In that cases the range of the data is duplicated. For instance, the square of a 16-bit fixed point value has a range of 32-bits. Thus a double precision integral image (with a 53-bit mantissa) can hold 2 million (or 2^{53-32}) of such 32-bit values without loss of precision, as a reference 2 million pixels equal to a 1400×1400 pixels image.

  Figure 5 illustrates the precision of the integral image evaluation for a large (2048×2048 pixels) synthetic image, for several different data types. The pixel values of this image are in the range $[2^{-19}, 2^{19}]$, which correspond to taking squares of data with a 19-bit precision. This can occur when computing L^2 distances between images with fixed point precision. Note how in Figure 5, for the single precision case, the error grows with the distance from $(0, 0)$. This is due to the accumulation of truncation errors.

- **Nightmare scenario.** The floating point data types are used to store all non integer values because of their flexibility. Ultimately, what matters for the integral image is the fixed point precision of its data type, namely the number of possible values the data can take in a certain range. The nightmare scenario for the integral image is to accumulate floating point data that spans an extreme range of floating point values (i.e. 2^{-100} to 2^{100}). Indeed, storing this data at fixed point precision requires more than 200 bits per pixel. Figure 6 shows an example of this scenario, with an image containing floating point values in the range $[2^{-19}, 2^{19}]$, but where we added 2^{40} to the pixels of the first 100 columns (saturated on the images) making the range of the whole image to become $2^{40+19} = 2^{69}$. For this case neither a single nor a double precision is sufficient to store the integral image without error. Note also that, although the first columns have the largest values in the images the precision there is reasonable, while it deteriorates over the portion of the image with smaller values. This is result of the truncation induced by the large values accumulated in the first band.

Although floating point data may easily surpass the integral image's precision, usual data do not span such large ranges of precisions, so one could still apply the integral image in most cases. As a rule of thumb, we can say that single precision integral images should NEVER be used, unless being very sure of not incurring in precision errors.

Preprocessing removing the average value. Removing the average value from the data increases the precision of the representation as seen in Figure 7. However, this improvement may be less important for some images. The cost of this pre-processing is one extra sweep through the data to compute the average, and one extra addition for each evaluation of the integral image.

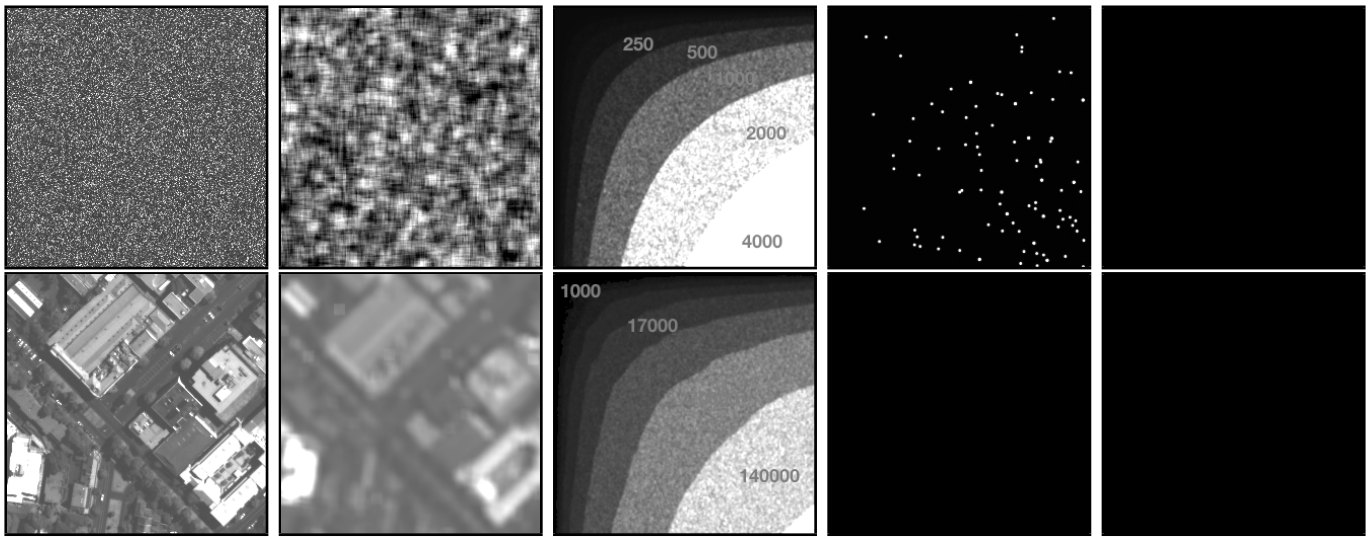

Figure 5: **Precision of the integral image for single, double and quadruple types.** In the first row from left to right, the first figure shows an extract from a 2048×2048 synthetic image with values in the range $[2^{-19}, 2^{19}]$. Such wide range of values corresponds to taking squares of data with a 19-bit precision, which can occur when computing L^2 distances between images with floating point data. The concrete image corresponds to squaring iid Gaussian noise. The second image shows the sum computed over patches of 11×11 pixels of the first image. The center image shows the error in the sums computed using a single precision integral image. The labels indicate the averaged error values. The error ranges from almost 0 (in the upper left corner) to 25000. The two remaining images show the error while using double and quadruple data types. In the first case the white points have a maximum error of 0.25, while in the case of quadruple precision the error is 0 everywhere.
In the second row the experiment is repeated with a natural image (of 2048×2048 pixels) with values in the range $[0, 2^{24}]$ (squared 12-bit integers). The second image shows the sum computed over patches of 11×11 pixels of the first image. The remaining images represent the errors in the sums computed using an integral image with single precision, double precision and quadruple precision respectively.

Figure 6: **Nightmare scenario for the precision of the integral image representation.** From left to right, the first figure shows an extract from a 2048×2048 image containing squared iid Gaussian noise with values in the range $[2^{-19}, 2^{19}]$, but where the pixels of the first 100 columns have been added to 2^{40} (and are therefore saturated on the image) making the range of the whole image to $[2^{-19}, 2^{40}]$. The second image depicts the sums computed over patches of 11×11 pixels of the first image. The first columns have values around 2^{46} (and are therefore saturated) while the remaining pixels are in the range $[0, 2^{25}]$. The center image shows the error in the sums computed using a single precision integral image. The labels indicate the average error values. in this case, all the errors are above 10^7. The two remaining images show the error for the double and quadruple data types. Note that although the largest values are in the first columns of the images the precision there is acceptable, while it deteriorates on the portion of the image with smaller values. This is due to the truncation induced by the large values that have been accumulated in the first band.

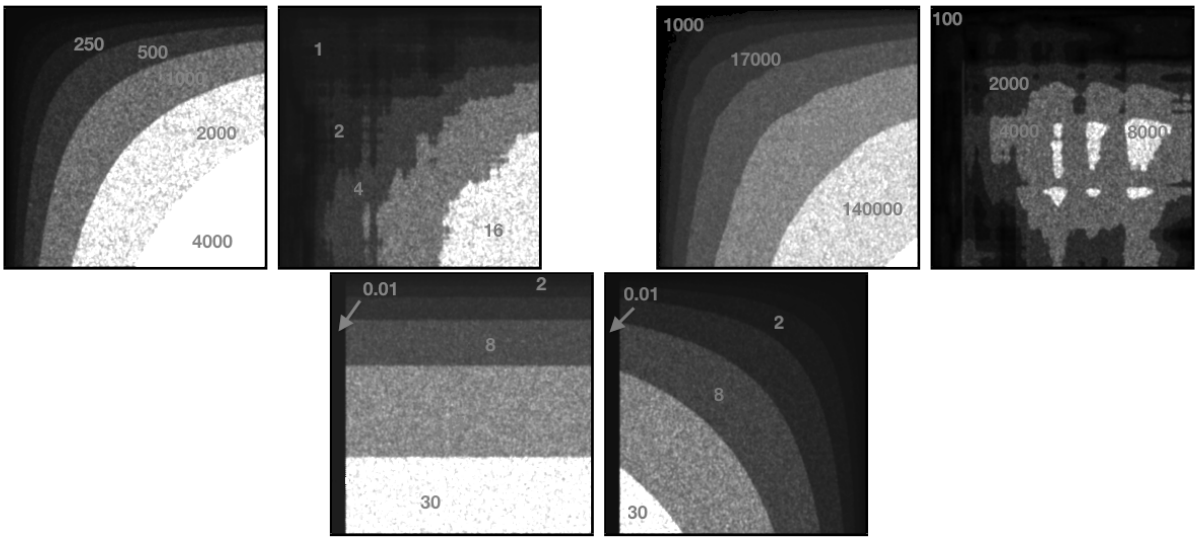

Figure 7: **Removing the average increases the precision.** The first pair of images shows the error of the single precision integral image from the synthetic experiment shown in Figure 5, and the error when the input data have been pre-processed by removing the average value of the image. This pre-processing increases the precision. The second pair corresponds to the natural image shown in Figure 5, on the left the error without pre-processing, on the right the error with pre-processing. In the third pair shows the comparison for the experiment of Figure 6 with double precision. In this case, although the error is more concentrated, its maximum value is the same.

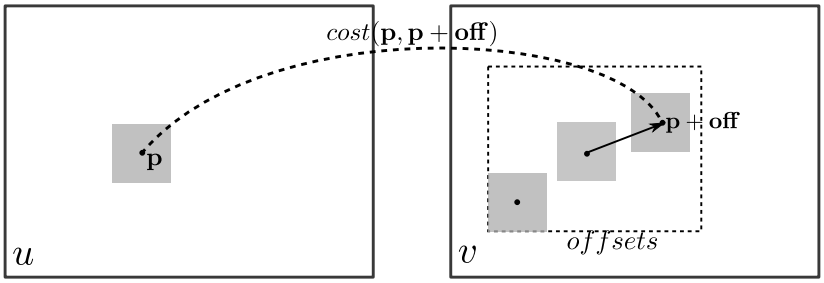

Figure 8: **Block matching.** The objective of block matching is to compute the matching costs between every patch of the reference image (centered at $\mathbf{p}$) and all the patches of the second image centered at $\mathbf{p} + \mathbf{off}$, where $\mathbf{off} \in offsets$.

3 Block Matching using Integral Images

In this section we describe an algorithm for efficiently performing block matching using integral images. We then specify the matching costs that can be implemented using integral images. Lastly we analyze the performance of this algorithm with respect to the exhaustive search and Fourier correlation for different window sizes.

The objective is to compute the matching costs between patches of two images (which may be the same image). For each pixel of the reference image, we want to compute the matching cost between the patch centered on this reference pixel and all the patches of the second image centered at pixels within a given range around the reference pixel, as illustrated in Figure 8. Let us denote $u, v : \Omega \to \mathbb{R}$ the reference and secondary images respectively, where $\Omega \subset \mathbb{Z}^2$ denotes the discrete image domain, and positions in $\mathbb{Z}^2$ are boldfaced, $\mathbf{p} = (x, y)$.

We consider matching costs that can be expressed as sums of distances between values at pixels $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$. For instance, taking the pixel value distance $d(x, y) = |x - y|^2$ summed over patches of size r^2 yields the matching cost known as SSD (Sum of Squared Differences)

$$\mathrm{SSD}(\mathbf{p}, \mathbf{q}) = \sum_{\mathbf{t} \in B(0,r)} d(u(\mathbf{p} + \mathbf{t}), v(\mathbf{q} + \mathbf{t})) = \sum_{\mathbf{t} \in B(0,r)} |u(\mathbf{p} + \mathbf{t}) - v(\mathbf{q} + \mathbf{t})|^2.$$

Supposing that the size of the images is $|\Omega| = N$ pixels, a naive evaluation of the matching costs for a set of M offsets entails (by the algorithm shown below) $O(NMr^2)$ operations.

```
1 for p ∈ u do
2     for off ∈ offsets do
3         cost ← 0;
4         for q ∈ B(0, r) do
5             cost ← cost + d(u(p + q), v(p + off + q);
```

Note, however, that if the evaluated patches are overlapping, therefore many of the computations are redundant. Particularly, if the pixels $\mathbf{p}$ are contiguous then each pixel is included in r^2 patches, so the same pixel value distance intervenes in the computation of r^2 matching costs.

The key of the integral image approach to block matching is to rewrite the previous algorithm so that each pixel value distance is computed only once, and reused in all of the r^2 patches, yielding a cost of $O(MN)$. This is shown in the algorithm below.

Concretely, note how the offset loop is placed outside the pixel loop. This implies that all the costs associated to a single offset are evaluated simultaneously. Then, for each offset all the pixel

```
for off ∈ offsets do
    for p ∈ u do
        dist(p) ← d(u(p), v(p + off))        /* precompute pixel distances          */
    ii ← ComputeIntegralImage(dist)          /* cost ∼ 2 additions per pixel        */
    for p ∈ u do
        cost ← SumPixelsEvaluatingIntegralImage(ii, p, r);
```

value differences are computed, and an integral image is produced from them, which in turn permits to compute the sums in constant time.

3.1 Matching Costs That Can Be Implemented with an Integral Image

Integral images can be used to compute any matching cost which can be expressed as sum of pixel values distances d, provided that these distances are independent of the pixel positions $\mathbf{p}$ and $\mathbf{q}$:

$$cost(\mathbf{p}, \mathbf{q}) := \sum_{\mathbf{t} \in B(0,r)} d(u(\mathbf{p}+\mathbf{t}), v(\mathbf{q}+\mathbf{t})).$$

Here we list some well-known costs that can be reduced to sums of independent pixel-wise distances, and therefore be efficiently implemented using integral images.

1. **SSD and SAD.** Are obtained by taking $d(a, b) = |a - b|^p$. Then for $p = \{1, 2\}$ we have the *Sum of Absolute Differences* (SAD) and the *Sum of Squared Differences* (SSD) respectively. But any value of p will work as well.

2. **ZSSD.** The *Zero-mean SSD* adds invariance with respect to additive contrast changes to the SSD. It is defined by

$$\text{ZSSD}(\mathbf{p}, \mathbf{q}) := \sum_{\mathbf{t} \in B(0,r)} \left| u(\mathbf{p}+\mathbf{t}) - v(\mathbf{q}+\mathbf{t}) + \mu_v(\mathbf{q}) - \mu_u(\mathbf{p}) \right|^2,$$

   where $\mu_v(\mathbf{q}) = \frac{1}{|B(0,r)|} \sum_{\mathbf{t} \in B(0,r)} v(\mathbf{q}+\mathbf{t})$ is the precomputed average of pixel values in v over the block centered at $\mathbf{q}$. The ZSSD cannot be implemented with integral images if expressed in this form, because the pixel value distances are dependent of the position of the pixels. Indeed, depending on the position, a different pair of means μ_v and μ_u are being subtracted. However, expanding the square we get

$$\text{ZSSD}(\mathbf{p}, \mathbf{q}) := \text{SSD}(\mathbf{p}, \mathbf{q}) - |B(0, r)| \left(\mu_v(\mathbf{q}) - \mu_u(\mathbf{p}) \right)^2$$

   and since $\mu_v(\cdot)$ and $\mu_u(\cdot)$ can be also precomputed by integral images, the overall cost of ZSSD is comparable to SSD.

3. **ZSAD [12] can't be implemented using integral images**, because the corresponding pixel distance $d(u(\mathbf{p}+\mathbf{t}), v(\mathbf{q}+\mathbf{t})) = |u(\mathbf{p}+\mathbf{t}) - v(\mathbf{q}+\mathbf{t}) + \mu_v(\mathbf{q}) - \mu_u(\mathbf{p})|$ is not independent of the current patch centers ($\mu_v(\mathbf{q})$ and $\mu_u(\mathbf{p})$ depend on the patch center).

   However, **a similar distance can be computed by pre-filtering the input images**. Following the naming introduced in the work by Hirschmuller and Scharstein [12] we call this method *mean filter*. It amounts to precompute $\tilde{u} = u - \overline{u}$ and $\tilde{v} = v - \overline{v}$, where $\overline{u} := u * \mathbf{1}_{B(0,r)}$. The cost is then computed using SAD taking $\tilde{u}$ and $\tilde{v}$ instead of u and v.

4. **SSDNorm.** Normalizing the patches by their L^2-norms renders the comparison robust to multiplicative contrast changes. The SSD/Norm is defined as

$$\mathrm{SSD/Norm}(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} \left| \frac{u(\mathbf{p}+\mathbf{t})}{\|u|_{B(\mathbf{p},r)}\|} - \frac{v(\mathbf{q}+\mathbf{t})}{\|v|_{B(\mathbf{q},r)}\|} \right|^2,$$

where $\|u|_{B(\mathbf{p},r)}\| = \sqrt{\sum_{\mathbf{t}\in B(0,r)} \left(u(\mathbf{p}+\mathbf{t})\right)^2}$. Expanding the above expression we get to a correlation formula

$$\mathrm{SSD/Norm}(\mathbf{p},\mathbf{q}) := 2 - 2\frac{\mathrm{Prod}(\mathbf{p},\mathbf{q})}{\|u|_{B(\mathbf{p},r)}\| \ \|v|_{B(\mathbf{q},r)}\|},$$

where $\mathrm{Prod}(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} u(\mathbf{p}+\mathbf{t})v(\mathbf{q}+\mathbf{t})$, which can be computed using integral images by taking $d(a,b) = ab$, and the terms $\|u|_{B(\mathbf{p},r)}\|$ and $\|v|_{B(\mathbf{q},r)}\|$ can be precomputed.

The implementation of this cost should prevent dividing by zero. Under this circumstance, the cost should be set to 2.

5. **NCC.** The *Normalized Cross Correlation* combines the benefits of ZSSD and SSDNorm as it is invariant to affine contrast changes. It is defined as

$$\mathrm{NCC}(\mathbf{p},\mathbf{q}) := 1 - \mathrm{Corr}(\mathbf{p},\mathbf{q}),$$

with

$$\mathrm{Corr}(\mathbf{p},\mathbf{q}) := \frac{\frac{1}{|B(0,r)|}\sum_{\mathbf{t}\in B(0,r)} \left(u(\mathbf{p}+\mathbf{t}) - \mu_u(\mathbf{p})\right)\left(v(\mathbf{q}+\mathbf{t}) - \mu_v(\mathbf{q})\right)}{\sqrt{\sigma^2\left(u|_{B(\mathbf{p},r)}\right)\ \sigma^2\left(v|_{B(\mathbf{q},r)}\right)}}, \tag{2}$$

and where $\sigma^2\left(u|_{B(\mathbf{p},r)}\right)$ is the sample variance of the block centered at $\mathbf{p}$.

Corr takes values in $(0,1]$, where 1 indicates the maximum correspondence. For a consistent notation across the different methods NCC is defined as $1 - \mathrm{Corr}$. The above expression can be written as

$$\mathrm{Corr}(\mathbf{p},\mathbf{q}) := \frac{\frac{1}{|B(0,r)|}\mathrm{Prod}(\mathbf{p},\mathbf{q}) - \mu_u(\mathbf{p})\ \cdot\ \mu_v(\mathbf{q})}{\sqrt{\sigma^2\left(u|_{B(\mathbf{p},r)}\right)\ \sigma^2\left(v|_{B(\mathbf{q},r)}\right)}},$$

which can be computed using one integral image (for Prod) for each offset value $(\mathbf{p}-\mathbf{q})$, and where the terms μ and σ^2 can be precomputed by integral images too!

The implementation of this cost must prevent dividing by zero. Under that circumstance the cost should be set to 1.

6. **AFF.** The *"affine" similarity measure* [5] is designed to distinguish patches independently of affine contrast changes, but unlike the NCC it can distinguish flat patches from those containing edges. It can be seen from (2) that if one of the patches is flat, then the correlation will be zero independently of the content of the second patch. In contrast, the *"affine" similarity measure* [5] defined below can be non zero under the same circumstances:

$$\mathrm{AFF}(\mathbf{p},\mathbf{q}) := \max\left(\min_{\alpha\geq 0,\beta} \|U_{\mathbf{p}} - \alpha V_{\mathbf{q}} - \beta\|\,,\ \min_{\alpha\geq 0,\beta} \|V_{\mathbf{q}} - \alpha U_{\mathbf{p}} - \beta\|\right),$$

where $U_{\mathbf{p}} = u|_{B(\mathbf{p},r)}$ and $V_{\mathbf{q}} = v|_{B(\mathbf{q},r)}$. It can be explicitly computed by the formula

$$\mathrm{AFF}^2(\mathbf{p},\mathbf{q}) := \max\left(\sigma^2\left(u|_{B(\mathbf{p},r)}\right), \sigma^2\left(v|_{B(\mathbf{q},r)}\right)\right) \cdot \min\left(1, 1 - \mathrm{Corr}(\mathbf{p},\mathbf{q})|\mathrm{Corr}(\mathbf{p},\mathbf{q})|\right),$$

which can be implemented with integral images after pre-computing $\sigma^2\left(u|_{B(\mathbf{p},r)}\right)$ and $\sigma^2\left(v|_{B(\mathbf{q},r)}\right)$, which in turn can be done with two integral images, one for the square of u and one for u.

Note that, unlike the NCC, the value of the *"affine" similarity measure* is not itself invariant to affine contrast changes. This implies that the cost associated to a pair of bright patches is higher than the cost associated to a pair of dark patches with the same structure as the bright pair.

LIN: is a simpler variant of the AFF cost that drops the invariance to additive changes, but has a similar performance

$$\mathrm{LIN}(\mathbf{p},\mathbf{q}) := \max\left(\min_{\alpha\geq 0}\|U_{\mathbf{p}} - \alpha V_{\mathbf{q}}\|, \min_{\alpha\geq 0}\|V_{\mathbf{q}} - \alpha U_{\mathbf{p}}\|\right).$$

It can also be implemented with integral images by re-writing it as

$$\mathrm{LIN}^2(\mathbf{p},\mathbf{q}) := \max(\|U_{\mathbf{p}}\|^2, \|V_{\mathbf{q}}\|^2)\left(1 - \frac{[\mathrm{Prod}(\mathbf{p},\mathbf{q})]^2}{\|U_{\mathbf{p}}\|^2\|V_{\mathbf{q}}\|^2}\right).$$

The implementation of these costs must prevent dividing by zero. Under that circumstance, the cost should be set respectively to the maximum variance or norm of the two patches.

7. **BTSAD and BTSSD.** By BTSAD or BTSSD, we mean an adaptation to SAD or SSD of the Birchfield and Tomasi [2] sampling insensitive pixel dissimilarity. This matching cost, originally proposed for stereo matching, is designed to be insensitive to image sampling. For smooth (non aliased) images this cost is proven to be stable with respect to subpixel translations of the patches, while still evaluating the costs at integer positions.

   This cost is particularly useful in combination with global block matching methods (Dynamic Programming for instance), where the dissimilarity is accumulated along scanlines, and where the subpixel computations are unaffordable. The insensitivity usually prevents the misclassification of some pixels as occlusions.

   The usefulness of this matching cost for a "local" block matching method is less clear, since the minimum SAD/SSD cost may not change. Nevertheless it is worth comparing its performance with subpixel matching.

   The matching costs proposed by Birchfield & Tomasi replace the definition of the pixel value distances with d^{BT}. For the case of the SSD we get

$$BTSSD(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} d^{BT}(I_L(\mathbf{p}+\mathbf{t}), I_R(\mathbf{q}+\mathbf{t})),$$

   where the distance d^{BT} is defined as a symmetrization of the distance by

$$d^{BT}(I_L(\mathbf{p}), I_R(\mathbf{q})) = \min(\bar{d}(I_L(\mathbf{p}), I_R(\mathbf{q})), \bar{d}(I_R(\mathbf{q}), I_L(\mathbf{p}))),$$

   and where $\bar{d}$ is

$$\bar{d}(I_L(\mathbf{p}), I_R(\mathbf{q})) = \max(0, I_L(\mathbf{p}) - I_R^{max}(\mathbf{q}), I_R^{min}(\mathbf{q}) - I_L(\mathbf{p})),$$

$$\bar{d}(I_R(\mathbf{q}), I_L(\mathbf{p})) = \max(0, I_R(\mathbf{q}) - I_L^{max}(\mathbf{p}), I_L^{min}(\mathbf{p}) - I_R(\mathbf{q})).$$

   The four precomputed images $I_R^{max}, I_R^{min}, I_L^{max}, I_L^{min}$ contain the maximum and minimum interpolated values of the image in a half pixel neighbor. For instance, with I_R we have

$$I_R^{min}(\mathbf{q}) = \min(I_R^-(\mathbf{q}), I_R^+(\mathbf{q}), I_R(\mathbf{q})) \quad \text{and} \quad I_R^{max}(\mathbf{q}) = \max(I_R^-(\mathbf{q}), I_R^+(\mathbf{q}), I_R(\mathbf{q})),$$

where the interpolated values are computed by bilinear interpolation:

$$I_R^-(\mathbf{q}) = \frac{1}{2}(I_R(\mathbf{q}) + I_R(\mathbf{q} - (1,0)^T)), \quad I_R^+(\mathbf{q}) = \frac{1}{2}(I_R(\mathbf{q}) + I_R(\mathbf{q} + (1,0)^T)).$$

Note that the maximum (and minimum) of the interpolated pixels I^{max} (resp. I^{min}) are only computed along the horizontal axis (definitions of I^+ and I^-). This is because the cost was originally proposed for stereo matching, so the subpixel differences are supposed to occur only along the horizontal axis. A straightforward extension of this cost for two dimensional block matching (**BT2DSAD** and **BT2DSSD**) replaces the definition of I^{max} (resp. I^{min}) to consider subpixel offsets also in the vertical direction

$$I^{max}(\mathbf{q}) = \max\left\{\hat{I}(\mathbf{q} + \mathbf{V})\right\} \quad \text{with} \quad \mathbf{V} = [-1/2, 1/2]^2,$$

where $\hat{I}$ denotes the bilinear interpolation of the image I. For this interpolation the maximum (resp. minimum) will occur at one of nine possible positions, therefore

$$I^{max}(\mathbf{q}) = \max\left\{\hat{I}(\mathbf{q} + \mathbf{s}) : s \in \{-\frac{1}{2}, 0, \frac{1}{2}\} \times \{-\frac{1}{2}, 0, \frac{1}{2}\}\right\}.$$

8. **Pre-filtering.** Hirschmuller and Scharstein in their work [12] describe a set of matching costs that amount to pre-filtering the input images. They consider: the *mean filter* which subtracts from each pixel the mean intensities within a fixed neighborhood of 15×15 pixels, the *Laplacian of Gaussian* which performs a smoothing that removes noise and intensity offsets, the *background subtraction by bilinear filtering* which smooths the image without removing high contrast textures working on windows of 15×15 pixels.

   Other types of pre-filters [12] such as the *rank filter*, *soft rank* and the *consensus filter*, cannot be implemented using integral images because they modify the pixel values on a patch basis.

It is important to remark that evaluating an integral image on a rectangle always reduces to a sum: $L_4 + L_1 - (L_2 + L_3)$, which is not guaranteed to be positive. To prevent the appearance of negative matching costs, due to the limited numerical precision, the implementation of the costs always truncate them at 0.

3.2 Block Matching Algorithm

We now detail the algorithm for performing block matching using integral images. As anticipated in the beginning of this section, the algorithm will evaluate all the matching costs for a given offset, one offset at a time (see Algorithm 3). For each offset, the costs of all the pixels $\mathbf{p}$ are computed, and at each position only the offset yielding the minimum cost is kept.

The same algorithm permits to compute the reverse matchings from the second to the reference image without overhead. Note that if the cost is symmetric, that is $cost(\mathbf{p}, \mathbf{q}) = cost(\mathbf{q}, \mathbf{p})$, and the considered offsets are all contiguous, then the costs computed for the reference image coincide with the costs from the secondary to the reference image (considering negative offsets). This is illustrated in Figure 9. This permits to compute the reverse matchings without overhead. Conversely, if the computed offsets are not contiguous, then some of the reverse matchings may not be considered.

Roughly, the block matching algorithm shown in Algorithm 3:

- First initializes the variables containing the minimum costs, and precomputes the images needed for evaluating the current cost. These images contain the patch averages, the patch variances, or patch norms, depending on the chosen cost.

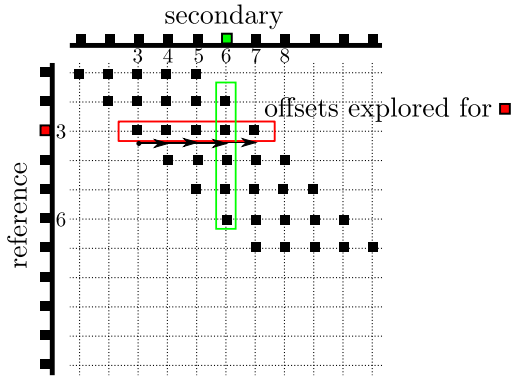

Figure 9: **1D illustration of matching and reverse matching.** The above figure shows a *disparity space diagram* representing all the possible matchings that can be computed from pixels in one line of the reference image and the corresponding line of the secondary image. The pixels of the reference image are represented in the vertical axis while the horizontal coordinates are the pixels of the secondary image. The dots on the grid represent the computed costs: for a pixel in the reference image several costs are computed corresponding to a set of 1D offsets. If the matching cost is symmetric, that is $cost(\mathbf{p}, \mathbf{q}) = cost(\mathbf{q}, \mathbf{p})$, then the computed costs also contain the matching costs from secondary to the reference image, for instance for the green pixel five negative offsets have been considered.

- For each offset, an image of pixel-wise distances is computed: $dist(\mathbf{p}) \leftarrow d(u(\mathbf{p}), v(\mathbf{p} + \mathbf{off}))$ Summing the values of *dist* over patches yields the matching costs for the current offset. For the computation of some costs it may also be necessary to access the precomputed images. The sums over patches is efficiently computed using integral images.
  - The integral image of *dist* is computed. Which permits to evaluate the matching cost of each patch in constant time.
  - For each pixel of the reference image, the cost for the current offset is computed. If the cost is smaller than any previous cost, then the offset is associated to the pixel. Likewise, the costs and the (reverse) matchings corresponding to the secondary image are updated as well.

A remark about parallelization. The previous algorithm can be parallelized by splitting the reference image into smaller sub-images and applying the algorithm in parallel to each sub-image. Besides reducing the computing time, this strategy also provides a way to deal with the numerical precision issues (mentioned in Section 2.3) connected to the use of large integral images. The block matching implementation associated to this work[2] process the reference image by horizontal bands, and each band is treated in parallel.

3.3 Subpixel Stereo Block Matching

The apparent motion of a point in an image due to camera displacement is related to its distance from the camera. This is the principle of stereoscopy, which allows to estimate the depths of a scene by determining the relative motion (parallax movement) of corresponding points in two images. In the pinhole camera geometry it can be shown that the parallax movement of a point occurs only along a line (called epipolar line), and that the images can always be *rectified* (transformed by an

[2] http://dev.ipol.im/~facciolo/ipol_demo/f_intimage_block_matching/

Algorithm 3: Block matching from image u to image v, and the reverse matching (v to u), using *integral images* for computing the matching costs.

input: The reference and secondary images: u and v. A patch size: r.
A set of offsets to evaluate: $offsets$.

output: The offsets of the matches from u to v and the corresponding minimum matching costs between the patches: matchOffset, minCost.
Offsets and costs for the *reverse* matches from v to u: R_matchOffset , R_minCost.

```
/* initialize the costs                                                    */
```
1 minCost $\leftarrow \infty$;
2 R_minCost $\leftarrow \infty$;
```
/* Precomputed images needed for the cost evaluation (μ_u, μ_v, σ_u,...)   */
```
3 Stuff $\leftarrow$ Precompute_Stuff(u, v) ;
4 **for off** $\in$ *offsets* **do**
```
    /* precompute pixel distances                                          */
```
5 **for** $\mathbf{p} \in u$ **do**
6 $dist(\mathbf{p}) \leftarrow d(u(\mathbf{p}), v(\mathbf{p} + \mathbf{off}))$
```
    /* compute integral image                                              */
```
7 $ii \leftarrow$ ComputeIntegralImage($dist$);
8 **for** $\mathbf{p} \in u$ **do**
```
       /* compute matching cost for pixel p and offset off by evaluating the
          integral image and the precomputed images                        */
```
9 newCost $\leftarrow$ EvalCostWithIntegralImage($ii, \mathbf{p}, \mathbf{off}$, Stuff);
```
       /* update the minimum costs                                         */
```
10 **if** *minCost(***p***)* $>$ *newCost* **then**
11 minCost($\mathbf{p}$) $\leftarrow$ newCost;
12 matchOffset($\mathbf{p}$) $\leftarrow$ **off**;
```
       /* update the Reverse matching                                      */
```
13 **if** *R_minCost(***p** $+$ **off***)* $>$ *R_newCost* **then**
14 R_minCost($\mathbf{p} + \mathbf{off}$) $\leftarrow$ R_newCost;
15 R_matchOffset($\mathbf{p} + \mathbf{off}$) $\leftarrow -\mathbf{off}$;

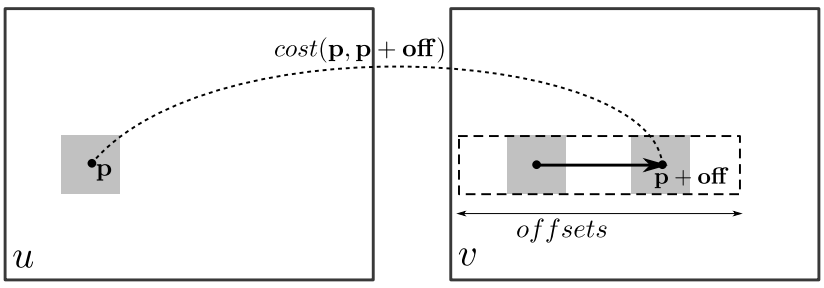

Figure 10: **Stereo Block matching.** For a stereo rectified image pair the possible correspondences are restricted to horizontal lines, therefore the stereo block matching only considers horizontal $offsets$.

homography) so that the apparent motion of pixels is horizontal everywhere [10]. The apparent motion on rectified images is often referred to as *disparity*.

The block matching algorithm described above can be used to estimate the disparity in rectified stereo pairs by restricting it to consider only horizontal offsets (see Figure 10). However, for estimating the depth it is useful to compute disparities with subpixel precision. Algorithm 4 extends the block matching for integer offsets to consider subpixel disparities. The principle is to compute several disparity maps with integer disparities, where in each pair one image is shifted by a subpixel quantity, and then combine the results in a single subpixel disparity map. To attain a precision of $1/n$-th of pixel for both the direct and reverse disparity maps, the integer precision algorithm must be invoked $2n$ times, meaning that the overall cost depends linearly on the precision $O(NnM)$ (where N is the number of pixels and M the number of offsets). The same subpixel procedure can be applied for estimating subpixel bi-dimensional offsets. However, the cost becomes $O(Nn^2M)$, which dims the value of the integral image with respect to other subpixel techniques for *optical flow* computation [13, 17].

Algorithm 4: Subpixel stereo block matching using integral image.

input: Images: u and v. A subpixel precision $s < 1$. A disparity range: $offsets$.
output: Disparities and minimum matching costs: matchOffset, minCost.
Reverse matches: R_matchOffset, R_minCost.

```
/* initialize the costs                                                    */
minCost ← ∞ ;
R_minCost ← ∞ ;
/* subpixel block matching:  u to v                                        */
for shift ∈ {ns : n ∈ ℕ, ns < 1} do
    shifted_u ← Shift_Horizontally(u, shift);
    [newOff,newCost,R_newOff,R_newCost] ← BlockMatching(shifted_u, v, offsets);
    for p ∈ u do
        if minCost(p) > newCost(p) then
            minCost(p) ← newCost(p);
            matchOffset(p) ← newOff(p) + shift;
/* subpixel block matching:  v to u                                        */
for shift ∈ {ns : n ∈ ℕ, ns < 1} do
    shifted_v ← Shift_Horizontally(v, shift);
    [newOff,newCost,R_newOff,R_newCost] ← BlockMatching(u, shifted_v, offsets);
    for p ∈ v do
        if R_minCost(p) > R_newCost(p) then
            R_minCost(p) ← R_newCost(p);
            R_matchOffset(p) ← R_newOff(p) + shift;
```

3.4 Comparison with Exhaustive Block Matching

As hinted in the introduction the performance of the exhaustive block matching decreases with the patch size. The results summarized in Table 1 corroborate this. We also note that for patches smaller than 3×3 the exhaustive search becomes more efficient than applying the integral image.

There are however many cases in which an exhaustive block matching is preferable to using the integral images.

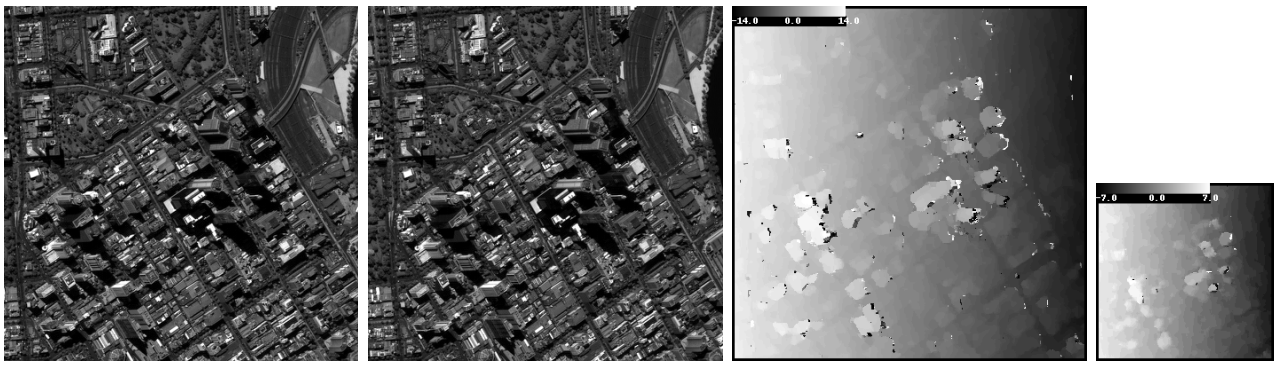

Figure 11: **Disparity range.** On the right is shown the disparity corresponding to the stereo pair on the left. The range of the offset spans a large set of disparities because of the "ground plane". This tendency could be determined at lower resolutions at a lower cost.

- **Variable offset ranges.** The block matching's computational cost depends linearly on the number of offsets being evaluated; for this reason it is always preferable to explore a small range of offsets. However, in many cases, although the overall offset range can be large, the range remains locally small. The stereo pair shown in Figure 11 illustrates this situation. In spite of the fact that the disparities induced by the structures are all relatively small, the disparity map is dominated by a global movement due to the change of view angle, which could be easily estimated at a lower resolution. This idea of progressively refining coarse approximations of the offset field is at the base of coarse-to-fine [18] techniques for block matching.

  The exhaustive block matching can be easily adapted to implement a coarse-to-fine refinement, by locally adapting the offset range. On the other hand the Algorithm 3, which uses the integral image, becomes very inefficient since each offset is tested for the entire image. A possible workaround for adapting the disparity range in Algorithm 3 consists in splitting the reference image in small overlapped tiles, and then applying the algorithm on each tile with a more reduced range. This strategy is not considered in this work.

- **Non overlapping patches.** A common trick to accelerate block matching is to compute the correspondences only on a coarse grid over the reference image. However, there is no benefit in using the integral image to compute the costs if the considered patches do not overlap. Indeed, when computing the correspondences for points on a grid with a stride bigger than the patch width, there is no gain in using the integral image, since the considered patches are all disjoint.

- **Matching costs based on feature vectors.** The block matching can be performed by comparing vectors of local features extracted from the image, instead of comparing the pixels in a patch. These features vectors are a high level description of the patch usually introducing some invariance.

  For this type of features the matching cannot be accelerated by using the integral image because the feature vectors of neighboring pixels do not overlap. The integral image may be useful, however, for computing the features themselves [1].

- **Matching costs using non uniform weights.** A drawback of block matching methods is their inability to distinguish distinct heights within the patch. This leads to incorrect estimations near the objects' boundaries and to the appearance of artifacts like the *foreground fattening*. The *adaptive weights* technique [25] prevents some of these issues by introducing weights in the computation of the matching costs. The weights are computed taking into account the image geometry in order to avoid considering more than one object in each patch.

Table 1: Performance comparison of block matching with integral image against the exhaustive search. The former is $O(MN)$ while the latter is $O(r^2MN)$, where M is the number of considered offsets, N is the number of pixels in the image and the side of the patch is r. Here $N = 512^2$ and $M = 100$.

Patch size (side r)	Integral image	Exhaustive search
11	6s	71s
9	6s	48s
7	6s	28s
5	6s	15s
3	6s	6s

This leads to improved results near the boundaries of the objects.

Although the weighted matching costs can be computed using integral images (see Section 2.2), their cost will be proportional to the complexity of the weight function (for weights $w(x, y)$ it is proportional to the cardinality of the set $\{(x, y) : \frac{d^2}{dx^2}w(x, y) \neq 0\}$). Usually the adaptive weights are incorporated as a refinement step operating over an initial coarse disparity map.

3.5 Comparison with Fourier Methods

The convolution theorem provides another way to accelerate the block matching. Any matching cost that can be expressed in terms of $\text{Prod}(\mathbf{p}, \mathbf{q})$ (including SSD) admits a re-writing as convolution of images

$$\text{Prod}(\mathbf{p}, \mathbf{x}) = (\tilde{u} * \tilde{v})(\mathbf{x}),$$

where $\tilde{u}(\mathbf{x}) = u(\mathbf{p} + \mathbf{x})$ and $\tilde{v}(\mathbf{x}) = \mathbf{1}_{B(0,r)}(\mathbf{x})v(-\mathbf{x})$. Applying the convolution theorem we have

$$\text{Prod}(\mathbf{p}, \mathbf{x}) = (\tilde{u} * \tilde{v})(\mathbf{x}) = \mathcal{F}^{-1}\{\mathcal{F}\{\tilde{u}\} \cdot \mathcal{F}\{\tilde{v}\}\}(\mathbf{x}).$$

This implies that, for a patch of size $r \times r$ of the reference image (centered at $\mathbf{p}$), all the matching costs within the search window can be computed at a cost proportional to the Fourier transform of the search region. Then, it is the size of the search region that determines the computational cost. Indeed, assuming that the M offsets to be evaluated are arranged in a squared region, and that $r^2 < M$, then the cost becomes approximately $O(M \log_2(M))$ (with a conservative constant of 30 due to the FFTs). We deduce that the Fourier transform computes the matching costs faster than the exhaustive search, as soon as $r^2 > 30 \log_2(M)$. This is the case for template matching [15], where the patches (r^2) and the search regions (M) are both large.

A key characteristic of template matching is that the set of features to be matched is usually small and sparse, be they points of interest or tracking points. In this case where there is no overlap between the features, the integral image will perform as the exhaustive search with $O(Mr^2)$ operations for each feature, while the Fourier method has an $O(M \log_2(M))$ cost per feature. Conversely, applying the Fourier method for computing the matches for all the pixels in the reference image is $O(NM \log_2(M))$, which is worse than using the integral image.

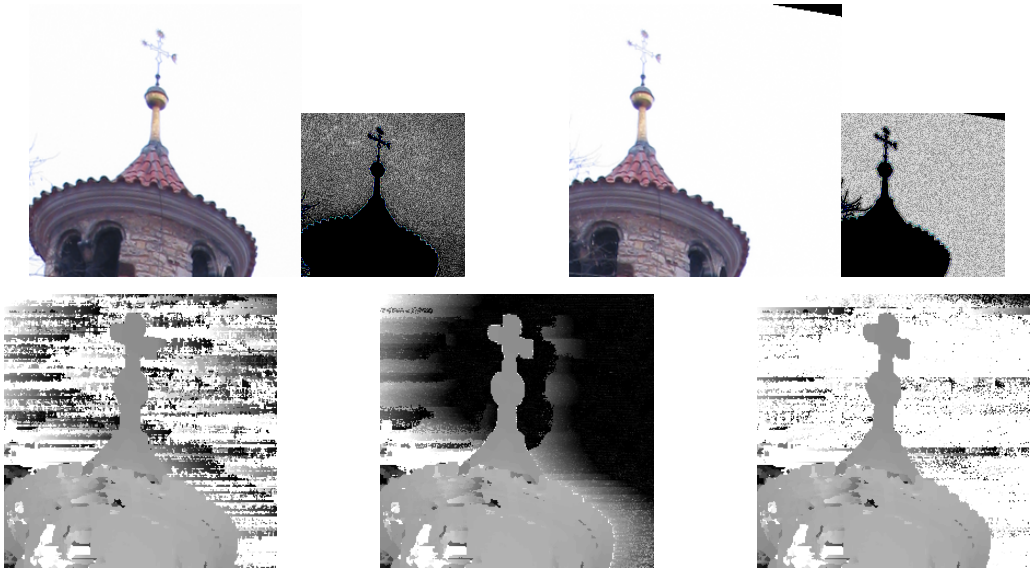

Figure 12: **Ghost artifacts in block matching.** The two images in the first row show a crop from a stereo pair. Although the sky looks saturated it is not uniform, the smaller images with increased contrast show the noisy values in the sky region. In the second row, the first image shows the disparity map obtained using exhaustive block matching applied to the stereo pair. As expected, the matches in the sky are erratic. The disparity map shown in the second image is computed using a single precision integral image. Note the appearance of a "ghost" of the main structure in the sky region, it is caused by the truncation errors in the integral image representation. Conversely, the result obtained using a double precision integral image (not shown) coincides with the one of the exhaustive matching. The last image shows the result obtained with a single precision integral image but processing the image by horizontal bands. Although the artifact have disappeared the result differs from the exact one, which indicates that some truncation errors are still present.

4 Limitations

In the previous sections we pointed out some limitations of using integral images for block matching, let us summarize them here:

- The **limited performance gain** with respect to exhaustive block matching on images with variable offset ranges, which are common in coarse-to-fine algorithms.
- There is **no performance gain** with respect to exhaustive block matching when computing the integral image on a coarse grid with non overlapping patches.
- Integral images **cannot be used to implement all matching costs**. For example, they do not adapt to ZSAD. Neither are they applicable to **matching costs based on feature vectors** or to computing costs with **adaptive weights**.
- The limited **precision of the data type used for representing the integral image** was remarked in Section 2.3. Let us illustrate how this issue affects the performance of block matching. Figure 12 shows a disparity map with a ghost artifact which is the product of the insufficient precision of the integral image representation. Indeed, the single precision float is insufficient in this case. There are two ways to deal with this issue, one by reducing the size of the integral image, and the other by involving a higher precision. A possibility to reduce the size of the integral image is to process the input images by bands. Figure 12 illustrates the results of both approaches.

5 Take Home Message

The integral image is a powerful tool for computer vision, that is also effective for accelerating block matching algorithms, as long as the matching cost can be expressed as sum of pixel-wise costs and the offset set to be tested is small. Its performance comes from a re-arrangement of the computations that capitalizes on patch overlap. Therefore, if the "query patches" do not overlap, it is more convenient to resort to Fourier based correlation, as it is the case for template matching. Moreover, for applications where the patches are encoded by feature vectors, there is no advantage in using the integral image. Lastly, beware of single precision integral images, as they may entail precision errors.

Image Credits

 © Jan Čech, crop from StMartin rectified stereo pair[3]

 © CNES, zoomed, cropped, satellite image of Melbourne 2012.

References

[1] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-Up robust features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008. http://dx.doi.org/10.1016/j.cviu.2007.09.014.

[2] S. Birchfield and C. Tomasi. A pixel dissimilarity measure that is insensitive to image sampling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(4):401–406, Apr. 1998. http://dx.doi.org/10.1109/34.677269.

[3] A. Buades, B. Coll, and J. M. Morel. A review of image denoising algorithms, with a new one. *SIAM Multiscale Modeling & Simulation*, 4(2):490–530, 2005. http://dx.doi.org/10.1137/040616024.

[4] F. C. Crow. Summed-area tables for texture mapping. *ACM SIGGRAPH Computer Graphics*, 18(3):207–212, July 1984. http://dx.doi.org/10.1145/964965.808600.

[5] J. Delon and A. Desolneux. Stabilization of Flicker-like effects in image sequences through local contrast correction. *SIAM Journal on Imaging Sciences*, 3(4):703–734, Oct. 2010. http://dx.doi.org/10.1137/090766371.

[6] K. Derpanis, E. Leung, and M. Sizintsev. Fast scale-space feature representations by generalized integral images. In *Proceedings of the 2007 International Conference on Image Processing*, volume 4, pages 521–524. IEEE, Oct. 2007. http://dx.doi.org/10.1109/ICIP.2007.4380069.

[7] O. Faugeras, T. Viéville, E. Theron, J. Vuillemin, B. Hotz, Z. Zhang, L. Moll, P. Bertin, H. Mathieu, P. Fua, G. Berry, and C. Proy. Real-time correlation-based stereo: algorithm, implementations and applications. Rapport de recherche RR-2013, INRIA, 1993. Available Online[4].

[8] L. A. Ferrari, P. V. Sankar, J. Sklansky, and S. Leeman. Efficient two-dimensional filters using B-spline functions. *Computer Vision, Graphics, and Image Processing*, 35(2):152–169, Aug. 1986. http://dx.doi.org/10.1016/0734-189X(86)90024-1.

[3] http://cmp.felk.cvut.cz/~cechj/GCS/stereo-images/StMartin/

[4] http://hal.inria.fr/inria-00074658

[9] L. A. Ferrari and J. Sklansky. A fast recursive algorithm for binary-valued two-dimensional filters. *Computer Vision, Graphics, and Image Processing*, 26(3):292–302, June 1984. http://dx.doi.org/10.1016/0734-189X(84)90214-7.

[10] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2nd edition, Apr. 2004. ISBN 0521540518.

[11] P. S. Heckbert. Filtering by repeated integration. *ACM SIGGRAPH Computer Graphics*, 20(4):315–321, Aug. 1986. http://dx.doi.org/10.1145/15886.15921.

[12] H. Hirschmuller and D. Scharstein. Evaluation of stereo matching costs on images with radiometric differences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(9):1582–1599, Sep. 2009. http://dx.doi.org/10.1109/TPAMI.2008.221.

[13] B. K. P. Horn and B. G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203, Aug. 1981. http://dx.doi.org/10.1016/0004-3702(81)90024-2.

[14] J. Jain and A. Jain. Displacement measurement and its application in interframe image coding. *IEEE Transactions on Communications*, 29(12):1799–1808, Dec 1981. http://dx.doi.org/10.1109/TCOM.1981.1094950.

[15] J. P. Lewis. Fast template matching. In *Vision Interface 95, Canadian Image Processing and Pattern Recognition Society*, pages 120–123, May 1995. Available Online[5].

[16] R. Lienhart and J. Maydt. An extended set of Haar-like features for rapid object detection. In *Proceedings of the 2002 International Conference on Image Processing*, volume 1, pages 900–903. IEEE, Sep. 2002. http://dx.doi.org/10.1109/ICIP.2002.1038171.

[17] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, volume 2, pages 674–679. Morgan Kaufmann Publishers Inc., Aug. 1981. Available Online[6].

[18] L. H. Quam. Readings in computer vision: issues, problems, principles, and paradigms. chapter Hierarchical warp stereo, pages 80–86. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. Available Online[7].

[19] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1):7–42, 2002.

[20] P. Y. Simard, L. Bottou, P. Haffner, and Y. LeCun. Boxlets: a fast convolution algorithm for signal processing and neural networks. In *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems 11*, pages 571–577. MIT Press, Nov. 1998. Available Online[8].

[21] O. Veksler. Fast variable window for stereo correspondence using integral images. In *Proceedings of the 2003 Conference on Computer Vision and Pattern Recognition*, volume 1, pages 556–561. IEEE, June 2003. http://dx.doi.org/10.1109/CVPR.2003.1211403.

[22] P. Viola and M. J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, May 2004. http://dx.doi.org/10.1023/B:VISI.0000013087.49260.fb.

[5] http://www.cipprs.org/papers/VI/VI1995/pp120-123-Lewis-1995.pdf
[6] http://ijcai.org/Past%20Proceedings/IJCAI-81-VOL-2/PDF/017.pdf
[7] http://www.dtic.mil/dtic/tr/fulltext/u2/a461877.pdf
[8] http://books.nips.cc/papers/files/nips11/0571.pdf

[23] J. Wang, Y. Guo, Y. Ying, Y. Liu, and Q. Peng. Fast non-local algorithm for image denoising. In *Proceedings of the 2006 International Conference on Image Processing*, pages 1429–1432. IEEE, Oct. 2006. http://dx.doi.org/10.1109/ICIP.2006.312698.

[24] L. Williams. Pyramidal parametrics. *ACM SIGGRAPH Computer Graphics*, 17(3):1–11, July 1983. http://dx.doi.org/10.1145/964967.801126.

[25] K.-J. Yoon and I. S. Kweon. Adaptive support-weight approach for correspondence search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):650–656, Apr. 2006. http://dx.doi.org/10.1109/TPAMI.2006.70.